Introduction to AI

Project 02: Logical Agent
Wumpus world
Group 5 - Report

Table of Contents

1. Group Introduction 3
2. Overview 3
    2.1 Purpose, Scope & Brief Description 3
    2.2 Wumpus World 3
    2.3 Construction 5
    2.4 Work assignment 5
    2.5 Completion rate for project requirements 6
3. Algorithm description 7
    3.1 Knowledge base 7
    3.2 The search 7
4. Implementation 8
    4.1 Class Program 8
    4.2 Class Agent 10
    4.3 Other files 11
5. Test Case 13
    5.1 Map 01 13
    5.2 Map 02 14
    5.3 Map 03 15
    5.4 Map 04 16
    5.5 Map 05 17

Project 02 - Report

1. Group Introduction

- This project was built, developed, and completed by a group of 4 members:
  - 22127273 - Phan Hải Minh - **Team Leader**.
  - 22127084 - Mai Đức Duy.
  - 22127130 - Nguyễn Trần Minh Hoàng.
  - 22127250 - Trần Thành Long.
- Group ID: 5

2. Overview

2.1 Purpose, Scope & Brief Description

- **Purpose:** Design and implement a logical agent that navigates through the Wumpus world, a partially-observable environment.
- **Scope & Brief Description:**
  - The Wumpus world presents the following key features:
    - The environment is an underground cave with a network of interconnected two-dimensional rooms.
    - A room may contain a deadly pit, signaled by a perceivable breeze, or a fatal Wumpus monster, detectable via a discernible stench:
      - The agent will die immediately when entering a room containing one of those harmful factors. No withdrawal is possible.
      - The percepts are available in the four-neighborhood of the room containing one of those harmful factors.
    - The agent has arrows to shoot in the direction he is facing.
    - There is one chest of gold, located somewhere in the cave.
    - Movement options: forward, turn left, or right by 90 degrees.
  - We need to play two roles:
    - **Program**: Set up a program to build the map.
    - **Agent**: Explore the Wumpus world and get the highest score possible for that world, using either Propositional Logic or First-Order Logic (or both).
  - Output information about the search, including the percepts at every room the agent enters, the updates in the knowledge base after each new percept, and the action decided upon by the agent. The program should also output the score of the agent.

2.2 Wumpus World

- **Description:** Our Wumpus World for actual implementation slightly differs from the original problem description. The environment's specification is as follows:
  - Grid Layout: The grid size has increased from 4x4 to 10x10.
  - Elements:
    - Agent: The agent moves around the grid trying to achieve its goals.
    - Wumpus: A monster that kills the agent if they end up in the same cell.
    - Gold: The agent's goal is to find and retrieve the gold.
    - Pits: Dangerous cells that the agent falls into and dies.
    - Poisonous Gas: Reduces the agent's health by 25% if entered.
    - Healing Potions: Restores the agent's health by 25% when using.

- Percepts:
  - Breeze: Indicates an adjacent cell contains a pit.
  - Stench: Indicates an adjacent cell contains the Wumpus.
  - Scream: Heard if the Wumpus is killed.
  - Whiff: Indicates an adjacent cell contains poisonous gas. o Glow: Indicates an adjacent cell contains a healing potion.
- Agent's Actions:
  - Move Forward: Moves to the next cell in the direction the agent is facing.
  - Turn Left/Right: Changes the agent's direction.
  - Grab: Picks up gold or healing potions if present in the cell.
  - Shoot: Fires an arrow in the direction the agent is facing to kill the Wumpus. Arrows can only hit targets in adjacent cells.
  - Climb: Exits the cave (used when the agent is in the starting position).
  - Heal: Uses a healing potion to restore health.
- Goals: The primary goal remains to find the gold and return to the starting position without dying with the highest score the agent can achieve.

- **Requirement(s) and Constraint(s):**
  - **The agent does not know the information on the whole map.**
  - There may be any number of pits and chests of gold in the world.
  - There is at least one Wumpus.
  - The agent carries an infinite number of arrows.
  - When the agent grabs the healing potion, the glow of that healing potion will disappear.
  - The poison gas will last forever. The agent can be poisoned multiple times in the same cell.
  - When the Wumpus dies. It will scream and you will know this information.
  - After Wumpus dies, the stench of that Wumpus will disappear (you should update the state of map after Wumpus dies), the game will end when one of the following two conditions occurs:
    - The agent dies (by wumpus, pit or 0% of health).
    - The agent climbs out of the cave (exit the cave).
  - The following activities can give the agent certain amounts of points:

Activity	Points
Pick up each chest of gold	+5000
Shoot an arrow	-1000
Killed by the Wumpus or Fall into a pit	-10000
Climb of the cave	+10
All Agent’s Action	-10

→ We need to determine the score that the Agent gets after completing the game.

2.3 Construction

- **Programming Language:** Python.
- **External libraries:**
  - tkinter (for GUI)
  - pysat - for SAT solving
- **Other built-in libraries:**
  - itertools
  - random
  - queue
- Including a GUI to visually describe the operation of the algorithms and Wumpus world map that are built and presented to address the requirements.
- **Program execution instructions**:
  - Open the folder stored source code (included in the submission file).
  - Run **main.py**.
  - Notice that due to an unknown bug in tkinter, the Agent's move annotation in the dialogue might not be properly shown on macOS if it is being set to dark mode.

2.4 Work assignment

STT	Description	Due Date	Responsibility	%Complete
1	Discuss the project requirements and divide the work among the team members.	29/07/2024	22127273 22127250	100%
2	Research and build the Wumpus world based on the information and requirements provided in the project.	04/08/2024	22127273 22127250 22127084 22127130	100%
3	Implement class **Program**	11/08/2024	22127273	100%
4	Implement class **Agent**	11/08/2024	22127273	100%
5	Construct & test GUI	11/08/2024	22127084 22127130	100%
6	Perform preliminary testing of the project to ensure the program operates stably, without conflicts or minor errors.	15/08/2024	22127084 22127130 22127250 22127273	100%
7	Generate some maps with different structures to test and discuss the entire program	18/08/2024	22127084 22127273	100%
8	Prepare the report document and complete it.	19/08/2024	22127250	100%
9	Submission	19/08/2024	22127273	100%

2.5 Completion rate for project requirements

STT	Requirements	Responsibility	%Complete
1	Finish the problem successfully.	22127273 22127250	100%
2	Graphical demonstration of each step of the running process. You can demo on the console screen or use any other graphical library.	22127084 22127130	100%
3	Generate at least 5 maps with different structures such as position and number of Pit, Gold and Wumpus.	22127084 22127273	100%
4	Report algorithms, and experiment with some reflection or comments.	22127084 22127250 22127273	100%

3. Algorithm description

3.1 Knowledge base

- The Knowledge base (KB) of an agent is built upon the information the agent has gathered while moving throughout the map, which are stored as boolean clauses. To derive whether a statement is true or false from the KB, we do the following:
    - We append the **negated** clause into the KB. If the statement is true then the clauses of the KB will **not** be boolean-satisfiable, false for otherwise.
    - To quickly solve the SAT problem, we use a powerful solver called the Glucose3 solver, which is implemented in the PySAT library of Python.
- Since the Wumpus world requires us to sometimes remove a clause from the current KB, which is not supported by the Glucose3 solver, an instance of the solver must be initiated every time a query is asked. Because the solver initiation is relatively slow and a lot (on average over 1000) of queries are made per move, this approach leads to the agent's poor reaction time. To optimize, we initiate the solver every add and remove request instead of when querying, and because there are only 9 different properties per cell there are only 18 initiations of the solver (9 for addition, 9 for removal) initiation process each move, which drastically improves the runtime of the agent.

3.2 The search

- We first define these terms:
    - **Certain(E, x, y):** The cell (x, y) is certain to have the element E based on the KB. This can simply be done by querying for the clause in the KB.
    - **Impossible(E, x, y):** The cell (x, y) is certain to not have the element E based on the KB. This can be done by querying for the negated clause in the KB.
    - **Possible(E, x, y):** The cell (x, y) **might be possible** to have the element E based on the KB. To do this first we check if it is **impossible** for E to occur in the cell or not. Since for this approach many unexplored cells will also be marked as possible, we additionally check that at least one of the adjacent cells of (x, y) is **certain** to have E's percept.
    - When we say it is **certain/impossible/possible** for a cell to have an element E, we are referring to **Certain/Impossible/Possible(E, x, y)**.
- We then generate a set of goal cells for the search, all of which must be **safe** and **unvisited by the agent in previous moves**. A safe cell is defined as a cell that is **impossible to have** the harmful elements, specifically Pit, Wumpus or Poisonous Gas. If the set of goal cells are empty, which means we have visited all safe cells, the goal is set to (0, 0) and the agent will pathfind back to the origin to climb out.
- A Uniform-cost search is deployed to find the path to one of the goals with maximal score with the logic to handle the score based on the rules. All unexplored cells' properties are determined by whether it is **possible** to have that property on the cell. After obtaining the move, the new cell is then added to the set of visited cells of the agent.
- If the previous search does not find a path, we relax our conditions. A new set of goal cells are generated and a new search is deployed. However, the condition for a cell to be safe this time is that it is **possible to not have** the harmful elements. All unexplored cells' properties are also now determined by whether it is **certain** to have that property on the cell. This search allows for the agent to have a cell to move to in cases where there are too many percepts surrounding the agent.

4. Implementation

4.1 Class Program

- **Requirement(s):**
  - This class is responsible for building the map, and reporting information about the elements in the cell.
- **Description:**
  - Input: The given map is represented by a matrix, which is stored in the input file. The input file format is described as follows:
    - The first line contains an integer n, which is the size of the map.
    - n next lines with each line represents a string:
      - If the room is empty, it is marked by a hyphen character.
      - If the room has some things or signals such as Wumpus(W), Pit(P), Breeze(B), Stench(S), Agent(A) or Gold(G). Between two adjacent rooms is separated by a dot (.).
    - Input only includes Wumpus(W), Pit(P), Agent(A) and Gold(G), Poisonous Gas(P_G), Healing Potions(H_P). We need to update information about Stench(S), Breeze(B), Whiff(W_H) and Glow(G_L) on the map based on input data.
  - Output: The map with all information.
  - The **Program.py** file includes the code and the presentation of the GUI and **Program** class.
- **Implementation:**
  - Libraries used:
    - **tkinter**: for creating a graphical user interface (GUI).
  - Imports and Initial Setup:
    - The program imports necessary libraries, including **tkinter** library for the GUI, **os** for file handling, and **Agent** from a module named **core** (will be presented in more detail in the following section).
    - It initializes the Program class, which manages the main components of the GUI application.
  - Attributes:
    - self.running: Tracks if the agent is active.
    - self.run_interval: Sets a 500 ms interval for agent actions.
    - self.health, self.score: Track the agent's health and score.
    - self.world_map: Stores the map of the environment.
    - self.N: Size of the map (10x10 grid).
    - self.logic_steps: A list to track logical steps or actions.
    - self.loaded_map_file: Keeps track of the loaded map file.
    - self.text_position: Used to manage text in the logic frame.
    - self.smoke_image, self.smoke_coverage: Manage smoke coverage on the map, initially covering all except the starting position.
    - self.agent_position, self.agent_direction: Track the agent's position and direction in the environment.
    - self.agent: Instance of the Agent class controlling agent logic.
    - self.agent_images: Dictionary holding images for the agent facing different directions.

- Methods:
  - __init__(self, root): Initializes the main application, setting up the GUI components, agent, and initial states.
  - create_menu_frame(self): Creates the menu frame with the "Load Map" button.
  - create_map_frame(self): Creates the map frame where the Wumpus world grid is displayed.
  - create_control_frame(self): Creates the control frame with buttons for controlling the agent, and labels for health and score.
  - create_logic_frame(self): Creates the logic frame to display the propositional logic steps.
  - sync_logic_frame_size(self): Synchronizes the size of the logic frame with the map canvas.
  - load_map(self): Loads the map from a file, resets the game state, and updates the display.
  - read_map(self, filename): Reads the map data from a file and returns the size and world map.
  - add_signals(self): Adds signals (stench, breeze, etc.) to the map based on the presence of hazards like Wumpus and pits.
  - display_map(self): Displays the Wumpus world grid, including the agent and various signals.
  - color_map(self, symbol): Maps different symbols to specific colors for display.
  - update_logic_frame(self, text_top): Updates the logic frame with new text, representing the agent's actions.
  - next_step(self): Performs the next step in the agent's decision-making process.
  - _move_agent_position(self, direction): Updates the agent's position based on the given direction.
  - get_percepts(self, position): Retrieves the percepts from the agent's current position.
  - _grab_item(self): Handles the agent grabbing items like gold or healing potions.
  - _shoot_arrow(self): Executes the action of the agent shooting an arrow to kill the Wumpus.
  - _climb_exit(self): Handles the agent's action of exiting the cave, concluding the game.
  - _heal(self): Heals the agent when it picks up a healing potion.
  - run_agent(self): Starts the automatic running of the agent.
  - pause_agent(self): Pauses the agent's automatic movement.
  - _auto_move(self): Automates the agent's movement and actions while the game is running.
  - execute_move(self, move): Executes the move decided by the agent and updates the game state accordingly.
  - write_output(self): Writes the sequence of logic steps to an output file.
  - _turn_left(self): Turns the agent to the left (counterclockwise).
  - _turn_right(self): Turns the agent to the right (clockwise).
  - remove_signals(self, x, y, signals): Removes signals (stench, breeze, etc.) from the map when certain conditions are met.

→ This **Program** class is a GUI-based simulation for the game “Wumpus world”, where an agent navigates through a grid-based world.

4.2 Class Agent

- **Description:**
  - The agent can only know the components of the cell where it is standing.
  - The agent must go step by step and call the program to get information about the cell it is standing on from the class **Program**. From there, it makes percepts to find the direction of movement and infer objects based on the available information.
  - The **agent.py** file defines the **Agent** class, which is responsible for controlling an agent's behavior in a grid-based environment. This environment involves challenges and hazards that the agent must navigate using logic and stored knowledge.
- **Implementation:**
  - Imports:
    - Modules **itertools**, **random**, and **PriorityQueue** are used for various functionalities like generating possible moves, managing randomness, and implementing a priority-based search algorithm.
    - The **Knowledge** and **Node** classes are imported from other modules, which manage the agent's knowledge base and nodes used in search algorithms.
  - Attributes:
    - self.size: Size of the grid (10x10).
    - self.KB: An instance of the Knowledge class, which stores and manages the agent's knowledge of the environment.
    - self.position, self.last_position: Track the current and previous positions of the agent.
    - self.direction: The agent's current facing direction.
    - self.score: Tracks the agent's score based on actions and events.
    - self.health: The agent's health, which decreases under certain conditions.
    - self.has_potion: Boolean flag indicating if the agent possesses a healing potion.
    - self.history: A list of actions the agent has taken.
    - self.visited: A set of visited grid cells.
  - Methods:
    - Private Methods:
      - __update(self, properties): Updates the agent's knowledge base based on perceived properties in the environment.
      - __KB_check(self): Checks and prints the knowledge base's status.
      - __safe(self, x, y, fail_hard=True): Determines if a grid cell is safe to move into.
      - __search(self, fail_hard=True): Searches for the best action or path to take based on the current state and knowledge.
      - __trace(self, predecessor, end): Traces back actions leading to a goal using a predecessor dictionary.
      - __take_action(self, action): Executes a given action and updates the agent's state accordingly.
    - Public Methods:
      - move(self, properties): Main method for determining and executing the next move based on the agent's perceptions.

→ The **Agent** class implements a basic decision-making process for navigating a grid environment. It uses a priority queue for exploring potential actions and states, updating its knowledge base based on perceptions, and selecting the best action to maximize health, score, and safety. The code suggests a complex environment with potential hazards like pits (P), wumpuses (W), and other dangers, and the agent must navigate it intelligently.

4.3 Other files

- **node.py**:
  - Defines a simple **Node** class used in the search algorithm within the agent's logic. This class helps in managing the state of the agent during its navigation through the environment.
  - This class is integral to the search process used by the agent. It encapsulates the state information and provides a way to prioritize exploration based on the agent's score, health, and potion status. This class is used within the priority queue to select the most promising paths during the agent's decision-making process.

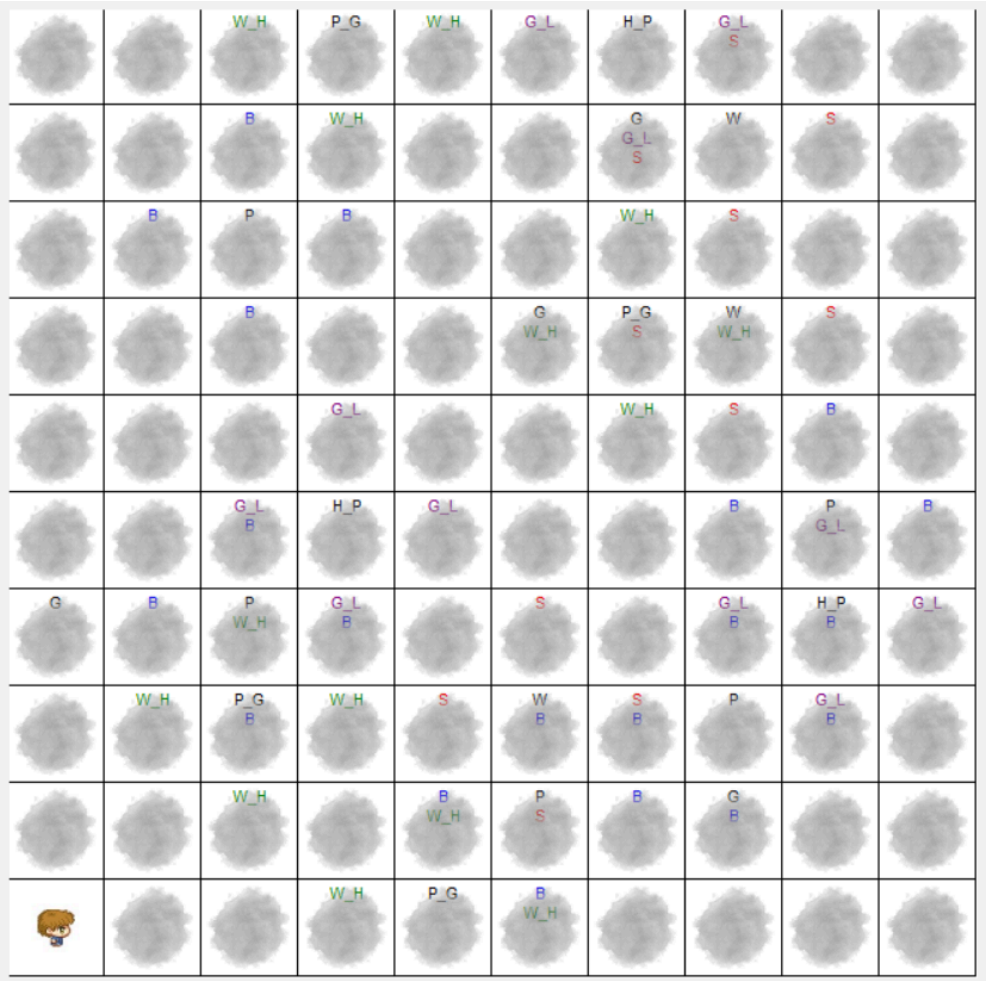
- **knowledge.py:**
  - Defines the **Knowledge** class, which is the agent's knowledge base, including the logical rules and constraints that dictate how the agent perceives and interacts with the environment.
  - Imports:
    - **itertools**: Used for generating combinations of grid coordinates.
    - **pysat.formula.CNF** and **pysat.solvers.Glucose3**: These modules from the PySAT library handle the creation and solving of Boolean formulas in Conjunctive Normal Form (CNF), enabling logical inference in the knowledge base.
  - Attributes:
    - self.size: The size of the grid.
    - self.percept: A dictionary mapping hazardous entities (Pits, Wumpuses, Poisonous Gasses) to their corresponding percepts (Breeze, Stench, Whizz).
    - self.__map: Maps symbols representing properties at specific coordinates to unique integers for CNF encoding.
    - self.__cache: Caches the results of previous queries to avoid redundant calculations.
    - self.__clauses: Stores the additional CNF clauses of the KB that the agent received from moving across the map.
    - self.__solver: A Glucose3 SAT solver which is initialized with the world's rules and additional CNF clauses from the environment.
  - Private Methods:
    - __adjacent(self, x, y): Returns a list of adjacent cells to a given (x, y) position within grid boundaries.
    - __symbol(self, name, x, y): Generates a unique symbol (integer) for a specific property at coordinates (x, y).
    - __set_rules(self): Returns a CNF clause that defines the initial rules of the environment in, such as the starting position being free of hazards, and how percepts relate to the presence of hazards in adjacent cells.
    - __reset_solver(self): Re-initialized the solver with the rules and the new clauses every time a clause is added or removed.

        - __query(self, clause): Checks if a specific clause (representing a property at a location) is entailed by the current knowledge base using the solver.
    - Public Methods:
        - add(self, property, x, y, existence=True): Adds a clause to the knowledge base, asserting the existence or non-existence of a property at a given location.
        - remove(self, property, x, y, existence=True): Removes a clause from the knowledge base, effectively retracting knowledge about a property at a location.
        - has(self, property, x, y, existence=True): Checks if a particular clause is already part of the knowledge base.
        - certain(self, property, x, y): Determines if the existence of a property at a specific location is certain.
        - impossible(self, property, x, y): Determines if the existence of a property at a specific location is impossible.
        - possible(self, property, x, y): Determines if the existence of a property at a specific location is possible.

→ The **Knowledg**e class is essential for reasoning about the agent's environment. It encodes logical rules and perceptions into a form that can be processed by a SAT solver, enabling the agent to make informed decisions about its actions based on what it knows and what it can infer. This class supports the agent in navigating complex environments by allowing it to determine the safety and likelihood of various hazards and rewards.

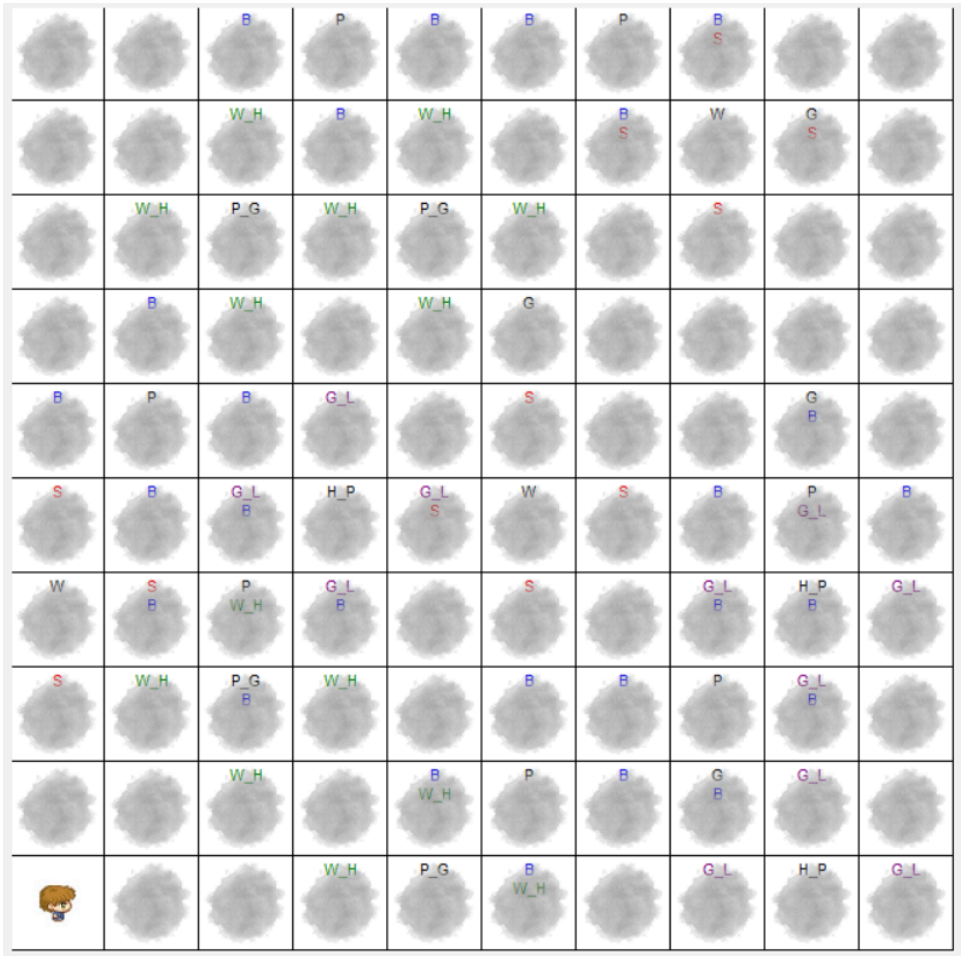
5. Test Case

5.1 Map 01



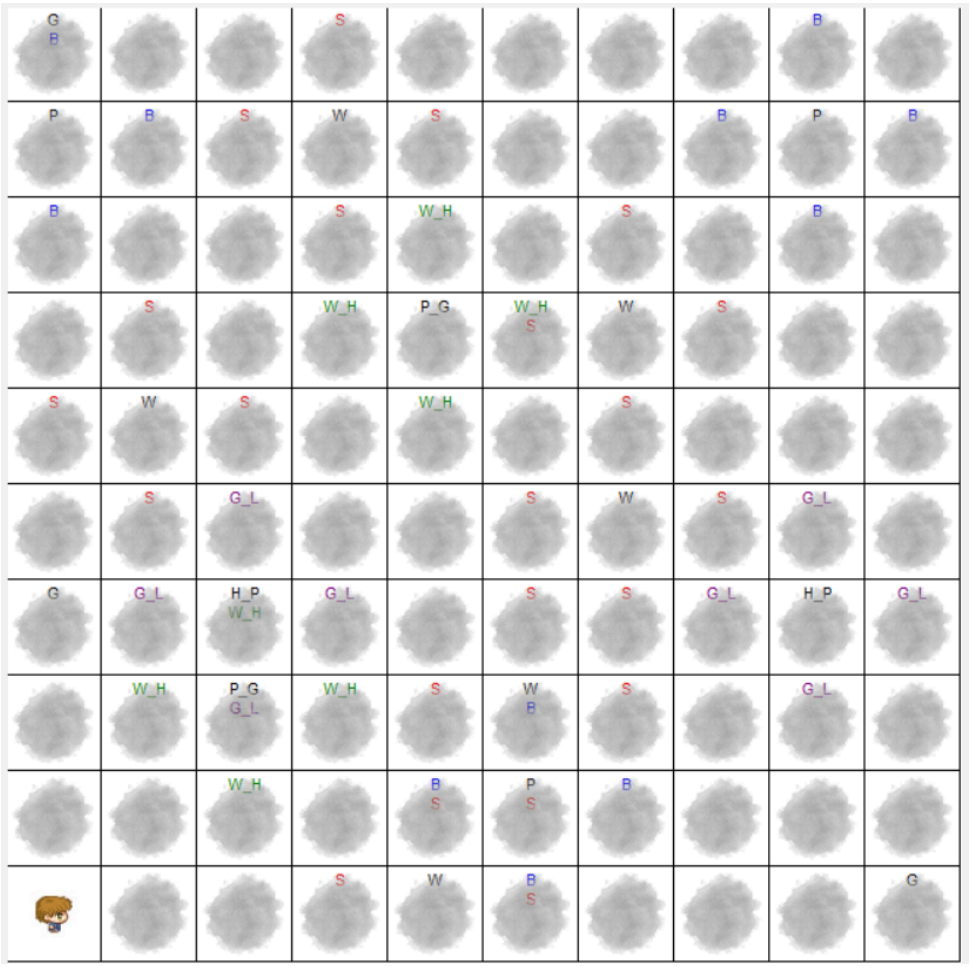
- This is the map for the first test case. It is stored in the file “input_1.txt” in the /testcase folder.
- The number of object in this map:
  - Wumpus: 3
  - Gold: 4
  - Pit: 5
  - Poisonous Gas: 4
  - Healing Potion: 3

5.2 Map 02



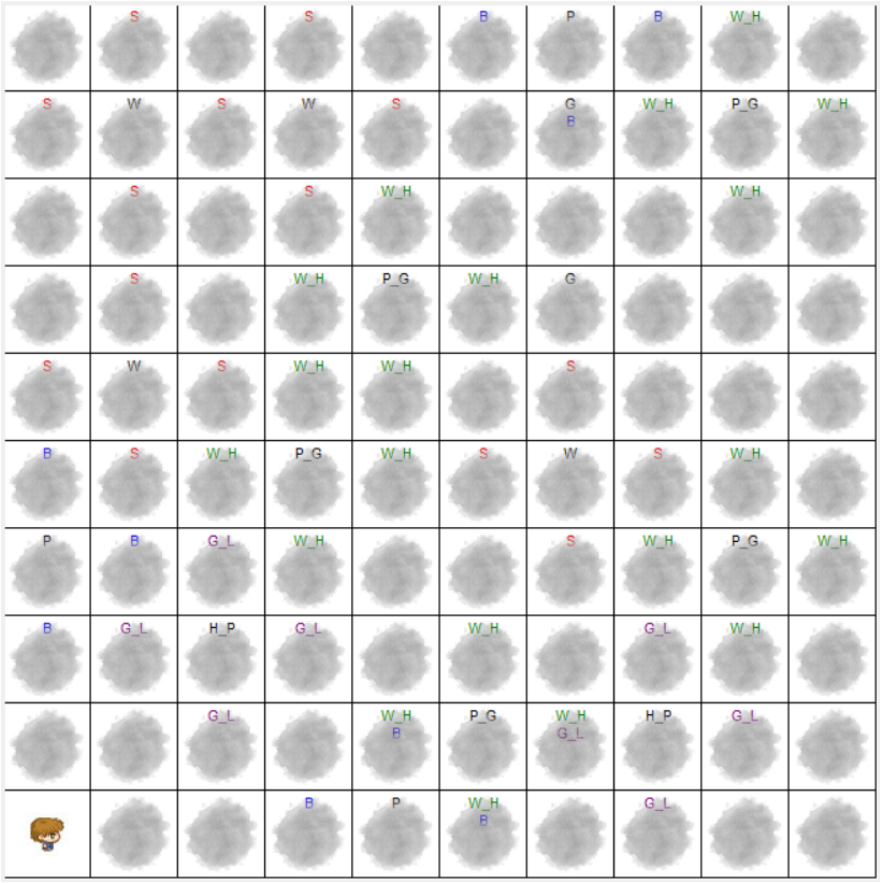
- This is the map for the second test case. It is stored in the file “input_2.txt” in the /testcase folder.
- The number of object in this map:
  - Wumpus: 3
  - Gold: 4
  - Pit: 7
  - Poisonous Gas: 4
  - Healing Potion: 3

5.3 Map 03



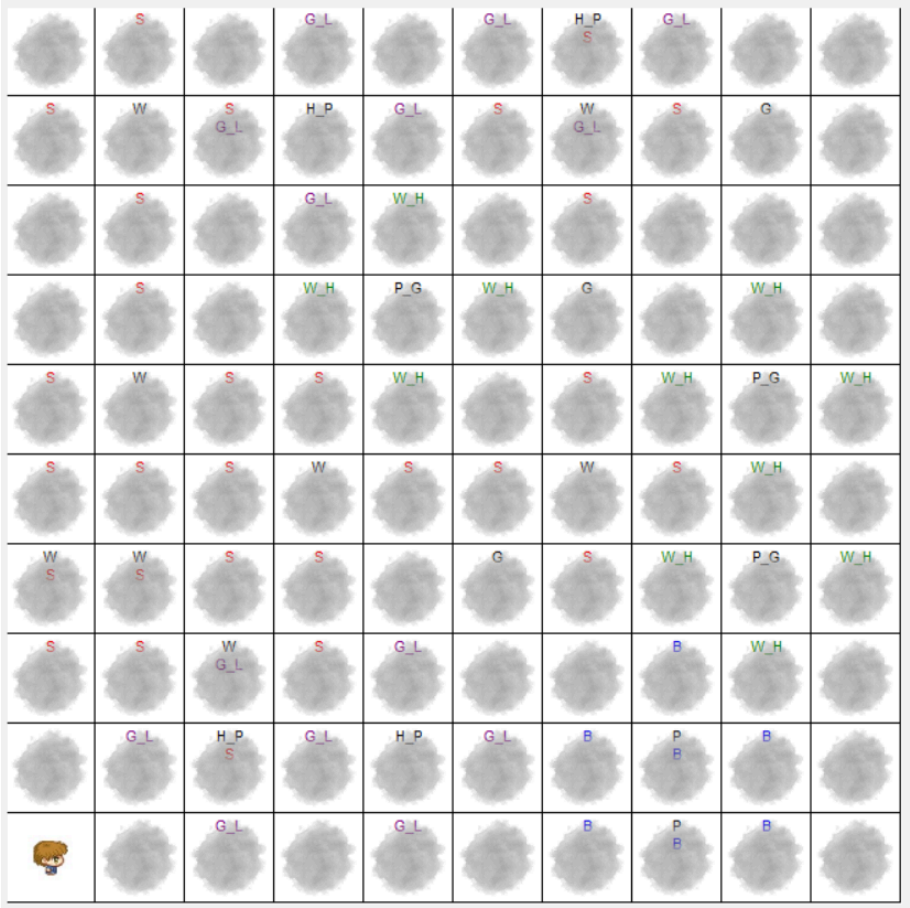
- This is the map for the third test case. It is stored in the file “input_3.txt” in the /testcase folder.
- The number of object in this map:
  - Wumpus: 6
  - Gold: 3
  - Pit: 3
  - Poisonous Gas: 2
  - Healing Potion: 2

5.4 Map 04



- This is the map for the fourth test case. It is stored in the file “input_4.txt” in the /testcase folder.
- The number of object in this map:
  - Wumpus: 4
  - Gold: 2
  - Pit: 3
  - Poisonous Gas: 5
  - Healing Potion: 2

5.5 Map 05



- This is the map for the fifth test case. It is stored in the file “input_5.txt” in the /testcase folder.
- The number of object in this map:
  - Wumpus: 8
  - Gold: 3
  - Pit: 2
  - Poisonous Gas: 3
  - Healing Potion: 4